Sublinear-Space Lexicographic Depth-First Search for Bounded Treewidth Graphs and Planar Graphs[†]

Taisuke Izumi[1]
Nagoya Institute of Technology, Japan
t-izumi@nitech.ac.jp

Yota Otachi
Nagoya University, Japan
otachi@nagoya-u.jp

Abstract

The lexicographic depth-first search (Lex-DFS) is one of the first basic graph problems studied in the context of space-efficient algorithms. It is shown independently by Asano et al. [ISAAC 2014] and Elmasry et al. [STACS 2015] that Lex-DFS admits polynomial-time algorithms that run with $O(n)$-bit working memory, where n is the number of vertices in the graph. Lex-DFS is known to be P-complete under logspace reduction, and giving or ruling out polynomial-time sublinear-space algorithms for Lex-DFS on general graphs is quite challenging. In this paper, we study Lex-DFS on graphs of bounded treewidth. We first show that given a tree decomposition of width $O(n^{1-\epsilon})$ with $\epsilon > 0$, Lex-DFS can be solved in sublinear space. We then complement this result by presenting a space-efficient algorithm that can compute, for $w \leq \sqrt{n}$, a tree decomposition of width $O(w\sqrt{n}\log n)$ or correctly decide that the graph has a treewidth more than w. This algorithm itself would be of independent interest as the first space-efficient algorithm for computing a tree decomposition of moderate (small but non-constant) width. By combining these results, we can show in particular that graphs of treewidth $O(n^{1/2-\epsilon})$ for some $\epsilon > 0$ admits a polynomial-time sublinear-space algorithm for Lex-DFS. We can also show that planar graphs admit a polynomial-time algorithm with $O(n^{1/2+\epsilon})$-bit working memory for Lex-DFS.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Theory of computation → Complexity classes; Theory of computation → Graph algorithms analysis; Mathematics of computing → Graph algorithms

Keywords and phrases depth-first search, space complexity, treewidth

Digital Object Identifier 10.4230/LIPIcs.ICALP.2020.67

Category Track A: Algorithms, Complexity and Games

Funding *Taisuke Izumi*: JSPS KAKENHI Grant Number 19K11824
Yota Otachi: JSPS KAKENHI Grant Numbers JP18H04091, JP18K11168, JP18K11169

1 Introduction

1.1 Background and Motivation

Depth-First Search (DFS) is one of the most fundamental and elementary graph search algorithms with a huge number of applications. Lexicographic DFS (Lex-DFS) is a popular variant of DFS, which requires the search head always moves to the *first* undiscovered neighbor in the adjacency list of the current vertex (as long as it exists). Recently, the space-efficient implementation of fundamental graph algorithms, including (Lex-)DFS, receives much attention [3, 6, 15, 16, 23, 26, 30]. These researches are roughly motivated by the

[1] Corresponding author.

two aspects as follows: First, the space matter is serious in the big-data (i.e., too large inputs) and/or IoT (i.e., too small computational devices) era. Second, the challenge of proving space-complexity lower bounds for problems within class P still lies at the core of computational complexity theory. One of the ultimate goals on this research direction is to prove or disprove the seminal P $\neq$ L conjecture [2]. We focus on the space complexity of Lex-DFS, particularly the algorithms using memory below the trivial $O(n \log n)$-bit bound. This can be motivated from both sides but much leans against the second one. The sublinear-space Lex-DFS problem is formulated as the one of outputting the Lex-DFS ordering of all vertices in streaming way, and its space complexity is measured by the required working-memory size, as the classical read-only model [25].

To argue the complexity of sublinear-space algorithms, the notion of P-completeness under logspace reduction plays an important role, which is analogous to NP-completeness in P $\neq$ NP conjecture. Reif [40] shows that Lex-DFS is P-complete under logspace reduction. It implies that no Lex-DFS algorithm only using $O(\log n)$-bit working memory exists unless P = L holds. A counterpart from the upper-bound side is recently obtained by a few literatures [3, 6, 23, 26]. They focus on the implementation of (Lex-)DFS achieving both polynomial time and $o(n \log n)$-bit space complexity, where n is the number of vertices in the input graph. Initiated by Asano et al. [3] and Elmasry et al. [23], a series of papers by several authors explore the time-space tradeoffs of (Lex-)DFS in the area of $o(n \log n)$-bit space-complexity. The state-of-the-art bounds are threefolds, $O(m \log^* n)$ running time and $O(n)$-bit working memory, $O(m + n)$ running time and $O(n \log \log(4 + m/n))$-bit working memory, and $O(m + n)$ running time and $O(n \log^{(k)} n)$-bit working memory for any integer $k > 1$, which are all proposed by Hagerup [26]. Looking at hidden coefficients, the smallest-space algorithm is the one by Asano et al. [3], which achieves a polynomial running time (with a large exponent) using the working memory of $n + o(n)$ bits. No algorithm so far attains cn-bit space complexity for $c < 1$, and obtaining such an algorithm is commonly recognized as a very challenging problem. This open problem is also supported from yet another context of computational complexity theory. Lex-DFS on directed graphs is at least as hard as the directed s-t reachability problem, which is known to be NL-complete and thus its space-efficient (ideally, logspace) solution is closely related to the seminal L = NL? problem. In fact, any directed Lex-DFS algorithm achieving $O(n^{1-\epsilon})$-bit space complexity for any small constant $\epsilon > 0$ would be a breakthrough result on this research line.

1.2 Our Result

In the context of directed s-t reachability, there are many attempts of attaining $O(n^{1-\epsilon})$-bit space complexity for a specific graph class such as planar or bounded treewidth graphs [1, 4, 5, 12, 13, 27, 29], which naturally yields the interest to the feasibility of sublinear-space Lex-DFS for those classes. It should be noted that Lex-DFS is P-complete even for planar graphs [2], and thus its difficulty under log-space solvability is the same as the general case. One of the main results presented in this paper is a sublinear-space Lex-DFS algorithm for bounded treewidth graphs. The first theorem is stated as follows.

▶ **Theorem 1.** *Let $0 < \epsilon < 1$ be an arbitrary positive constant, G be any n-vertex directed graph of treewidth w, and $(\mathcal{T}, \{B_x\}_{x \in V_\mathcal{T}})$ be its tree decomposition of width $w' \geq w$, where $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ is a tree and each node x in $\mathcal{T}$ is associated with a subset B_x of vertices in G. Assume a polynomial-time algorithm Alg enumerating the vertices in B_x for all $x \in V_\mathcal{T}$ and the edges in $E_\mathcal{T}$. Then there exists a Lex-DFS algorithm of running time $O(n^{O(1/\epsilon)})$ using $O(\epsilon^{-1} w' n^\epsilon \log n)$-bit memory (except for the space used by Alg).*

[2] L is the class of problems decidable with $O(\log n)$-bit working space (and thus in poly(n) time).

It should be noted that Lex-DFS does not necessarily lie on the seminal framework known as Courcelle's theorem [19] and its logspace version [22] because the output depends on the order of vertices in the adjacency list of the input graph.

To figure out a "purely" sublinear-space Lex-DFS algorithm, it is necessary to implement a tree decomposition algorithm using only sublinear space. Elberfeld et al. [22] presents a logspace tree-decomposition algorithm for $w = O(1)$, but no sublinear-space algorithm covering the case of $w = \omega(1)$ has been known so far. Our second theorem provides a sublinear-space solution for tree decomposition:

▶ **Theorem 2.** *There exists an algorithm that, given a graph G of n vertices and $w \leq n^{1/2}$, either provides a tree decomposition of width $O(wn^{1/2}\log n)$ or correctly decides that the treewidth of G is more than w. This algorithm runs in a polynomial time and uses $O(wn^{1/2}\log^2 n)$-bit space.*

To the best of our knowledge, this is the first non-trivial tree-decomposition algorithm attaining both sublinear-space and polynomial time for $w = \omega(1)$. It is also worth noting that planar graphs admit a space-efficient algorithm of finding a balanced separator of size $O(\sqrt{n})$ using $\tilde{O}(\sqrt{n})$-bit space [27], which can be translated into a small-space tree decomposition algorithm of width $O(\sqrt{n}\log n)$. Putting all the results above together, we obtain the following consequence:

▶ **Corollary 3.** *Let $\epsilon > 0$ be any positive constant. There exist the polynomial-time Lex-DFS algorithms respectively satisfying the following properties*[3]*.*

- *Using $O(n^{\epsilon})$-bit working memory for directed graphs of treewidth $w = O(1)$.*
- *Using $O(wn^{1/2+\epsilon})$-bit working memory for directed graphs of treewidth $w = O(n^{1/2})$.*
- *Using $O(n^{1/2+\epsilon})$-bit working memory for directed planar graphs.*

1.3 Related Work

As stated above, the space complexity of Lex-DFS is one of the classical problems in the context of logspace computability. Following the P-completeness result by Reif [40], Anderson and Mayr [2] also shows a weaker variant of Lex-DFS (lexicographically first maximal path) is also P-complete even for planar graphs. The main interest of those earlier results is closely related to the s-t reachability problem. It is known that the space complexity of undirected s-t connectivity drops into $O(\log n)$ bits for any input graphs, which is proved in Reingold's celebrating paper [41]. The best space upper bound of all polynomial-time directed s-t reachability algorithms is $O(n/2^{\Omega(\sqrt{\log n})})$ bits by Barnes et al. [9]. Its near optimality within a (naturally) restricted class of algorithms, called NNJAG [39], is also shown by Edmonds et al. [21].

More recently, the space-complexity matter of the directed s-t reachability problem for specific graph classes receives much attention, and a number of papers try to expand the graph class allowing sublinear-space directed reachability algorithms. Grid graphs [1, 5, 28], planar graphs [1, 4, 13, 27], bounded-genus graphs [12], and bounded-treewidth graphs [29] have been considered so far. Notice that the algorithms presented in [12] and [29] for bounded-genus graphs and bounded-treewidth graphs respectively require the surface embedding and the tree decomposition of the input graph (as Theorem 1), but it is not addressed how to compute them using sublinear space. Our tree-decomposition algorithm (by Theorem 2) yields the first

[3] Since one can choose an arbitrary $\epsilon > 0$, polylog(n) factors are absorbed in the part of n^{ϵ} in the statements of this corollary.

purely sublinear-space directed reachability algorithm for graphs of treewidth $w = O(n^{1/2-\epsilon})$. Very recently, an $O(1)$-approximate tree decomposition algorithm using $O(wn)$-bit space is presented [31], which attains a non-trivial space complexity for $w = o(\log n)$.

Despite relatively rich literatures on directed s-t reachability, the space complexity of Lex-DFS receives less attention until recently. After the two concurrent results by Asano et al. [3] and Elmasry et al. [23], a few follow-up papers propose fundamental graph algorithms using only $o(n \log n)$-bit working memory, which cover Lex-BFS [6, 23], single-source shortest path [23], biconnected component decomposition [15, 30], s-t numbering [15], maximum cardinality search [16], and so on. It is also becoming active to consider sublinear-space algorithms for fundamental non-graph problems [7, 20, 34, 36, 42].

On the side of computational models, the read-only model is one of the classical models to consider the complexity of working memory. An earlier topic in this model is the time-space tradeoffs for sorting and/or selection [10, 17, 18, 25, 37, 38]. Recently, more unconventional models are also investigated; Stream model [33], restore model (algorithms can manipulate input memory but after the computation the initial input data must be recovered) [14, 32], and catalytic model (algorithms can use a large memory which are already used for other purpose, and after the computation the memory state must be recovered to the initial one) [11]. Some of the results in those models allow a Lex-DFS algorithm using only a small (exclusive) working memory, but they are incomparable to the ones in the standard read-only model. Barba et al. [8] provides a general scheme of realizing stack machines using only a small memory space. While the dominant part of the memory usage in Lex-DFS algorithms is the storage for a stack, the technique by Barba et al. only applies to the algorithms whose access pattern to stacks are non-adaptive. Thus that scheme does not work for saving the space complexity of Lex-DFS algorithms.

1.4 Organization of Paper

In Section 2, we introduce the model, notations, and several auxiliary matters for our Lex-DFS problem. Sections 3 and 4 respectively show the proofs of Theorem 1 and 2. Finally the paper is concluded in Section 5.

2 Preliminaries

2.1 Model and Notation

As stated in the introduction, this paper adopts the *read-only model* [25], where the space complexity of an algorithm is measured by the number of bits used for the working space, excluding the memory for inputs and outputs. The input memory is read-only, and the output memory is write-only. The memory-access model follows the standard RAM of $(\log n)$-bit words. Let G be any directed input graph of n vertices and m edges, which is stored in the form of the adjacency list A_G. For any graph G, we denote the sets of the vertices and edges in G by V_G and E_G respectively. We assume $V_G = [0, n-1]$, that is, each vertex in V_G is uniquely identified by an integer in $[0, n-1]$. Letting $N_G(v) \subseteq V_G$ be the set of v's neighbors in G, we refer to the neighbor list of $v \in V_G$ as $A_{G,v}$ and denote the i-th vertex in $A_{G,v}$ by $A_{G,v}[i]$ (index i starts from value zero). We use notation $u <_v u'$ for $u, u' \in N_G(v)$ if u precedes u' in $A_{G,v}$. We define the inverse mapping of $A_{G,v}$ as $A^{-1}_{G,v}$, that is, for any neighbor u of v, $A^{-1}_{G,v}[u]$ returns the position of u in $A_{G,v}$. When we consider a subgraph $H \subseteq G$, the adjacency list of H is inherited from that of G. More precisely, when we delete an edge $(u, v) \in E_G$, the adjacency list $A_{H,u}$ after deletion is defined as the one such that

$A_{H,u}[i] = A_{G,u}[i]$ for any $i < A^{-1}_{G,u}[v]$ and $A_{H,u}[i-1] = A_{G,u}[i]$ for any $i > A^{-1}_{G,u}[v]$. Since any subgraph is obtained by iterative deletion of edges (and removal of isolated vertices), the specification of the adjacency list after deletion of one edge also specifies the adjacency list A_H for any subgraph H.

Letting X be a set of vertices or edges, we denote by $G - X$ the graph obtained from G by removing all the elements in X (if X is a vertex set, all the edges incident to a vertex in X are also deleted). The subgraph of G induced by X is denoted by $G[X]$. If there is a polynomial-time algorithm *Alg* enumerating all the elements in X, it naturally provides an access to the adjacency list A_H for $H = G[X]$ or $H = G - X$ without explicitly constructing it in the working memory (i.e., *Alg* works as a filter extracting the elements in $G[X]$ or $G - X$ from the adjacency list of A_G). Then we call *Alg* an *emulator* of H. The overhead of accessing to A_H is a polynomial time (depending on the running time of *Alg*) per one access. Thus we can run any polynomial-time algorithm taking H as its input within a polynomial time. In the following argument, we often omit the subscript G of the notations defined above if there is no ambiguity.

2.2 Lex-DFS

In what follows, we fix a starting vertex s of Lex-DFS tasks. A Lex-DFS algorithm is presented in Algorithm 1. In the algorithm, we introduce the notion of time. At each time, the search head v_{cur} moves to a neighbor decided by the algorithm. The search starts at time $t = 1$ and finishes at time $t = 2n$. We define h_t as the vertex pointed by the search head at the beginning of time t in the Lex-DFS on G. For vertex $v \in V$, the *discovery time* $d(v)$ of v is defined as the first time when the search head moves to v. Similarly, we define the *leaving time* $l(v)$ of v as the last time when the search head moves from v. The discovery time of s is defined as zero. Following the terminology in [3], a vertex v is called a *gray* vertex at t if $d(v) \leq t \leq l(v)$ holds. The (path) subgraph corresponding to the sequence of all gray vertices at time t sorted by their discovery times is called the *gray path* at t, which is denoted by S_t[4]. For any vertex $u \in V_{S_t}$, we also denote by $p_t(u)$ and $s_t(u)$ the (immediate) predecessor and successor of u in S_t, and by $S_t(u)$ the prefix of S_t terminating at u. We define $p_t(s) = \perp$ and $s_t(h_t) = \perp$.

In Algorithm 1, we encapsulate the space-consuming parts of the algorithm by two abstract procedures called Pivot(t) and Parent(t). The procedure Pivot(t) tries to find the first undiscovered neighbor of h_t with respect to the order $<_{h_t}$. If there is no undiscovered neighbor, it returns -1. The procedure Parent(t) returns the predecessor $p_t(h_t)$ in the current gray path. It returns -1 if $h_t = s$ holds. It is obvious that Lex-DFS is implemented with the working memory of $f(n) + O(\log n)$ bits if both of Pivot(t) and Parent(t) are implemented with $f(n)$ bits.

2.3 Tree Decomposition and Balanced Separator

We first present the definition of tree decomposition.

▶ **Definition 4.** *A tree decomposition of an undirected graph G is a pair $(\mathcal{T}, \{B_x\}_{x \in V_\mathcal{T}})$, where $\mathcal{T}$ is a tree and each node $x \in V_\mathcal{T}$ is associated with a subset B_x of vertices in V_G (called the bag x) satisfying the following conditions:*

- *Any edge in G is covered by at least one bag, i.e., $\forall (u, v) \in E_G : \exists x : u, v \in B_x$.*
- *Letting $\mathcal{T}(u)$ be the subgraph of $\mathcal{T}$ induced by the bags containing u, for any $u \in V_G$, $\mathcal{T}(u)$ is non-empty and connected.*

[4] Intuitively, the gray path S_t is the path from s to h_t in the Lex-DFS tree of G.

Algorithm 1 Lex-DFS Algorithm for graph G starting from s.

```
1: v_cur ← s; t ← 1                                   ▷ v_cur is the search head
2: while true do
3:     v ← Pivot(t)          ▷ Find the first undiscovered neighbor of v_cur in A_{G,s}.
4:     if v = −1 then                  ▷ All neighbors have been already visited
5:         v ← Parent(t)                          ▷ Find the parent in the gray path
6:         if v = −1 then halt                    ▷ All vertices are visited
7:     else
8:         Output v                               ▷ Discovery of a new vertex
9:     v_cur ← v
10:    t ← t + 1
```

Note that tree decomposition is defined for undirected graphs. When we consider the tree decomposition of directed graphs G, we naturally adapt the same definition to the undirected graph obtained from G by omitting the orientation of all edges[5]. Each bag is identified by an integer value in $[0, |V_{\mathcal{T}}| - 1]$. Throughout this paper, we assume that any decomposition tree $\mathcal{T}$ is rooted, and that a tree decomposition is encoded as the sequence $(B_1, q(1)), (B_2, q(2)), \ldots, (B_x, q(x)), \ldots, (B_{|V_{\mathcal{T}}|-1}, q(|V_{\mathcal{T}}| - 1))$, where $q(x)$ is the ID of the parent of x in $\mathcal{T}$. The parent of the root bag is defined as -1. A sublinear-space tree decomposition algorithm must output this sequence in a streaming way. The *width* w of a tree decomposition $(\mathcal{T}, \{B_x\}_{x \in V_{\mathcal{T}}})$ is defined as the maximum bag size minus one, i.e., $w = (\max_{x \in V_{\mathcal{T}}} |B_x|) - 1$. The *treewidth* of a graph G is the minimum width over all tree decompositions of G. It is a fundamental property that the removal of the vertices in any (non-leaf) bag B_x from G splits G into several connected components, each of which corresponds to a subtree of $\mathcal{T}$ obtained by the removal of x from $\mathcal{T}$.

Tree decomposition is closely related to the notion of balanced vertex separators. Let $G = (V, E)$ be any directed graph and $\mu \colon V_G \to \mathbb{N}$ be any vertex-weight function. We define $\mu(X) = \sum_{v \in X} \mu(v)$ for any $X \subseteq V_G$. A vertex subset $U \subseteq V$ is called a *weighted α-balanced separator of G with respect to μ* if any weakly-connected component C in $G - U$ satisfies $\mu(V_C)/\mu(V_G) \leq \alpha$. If μ is a constant function, it is simply called an *α-balanced separator of G*. Throughout this paper, we often consider a subgraph obtained by recursively removing separators. Let $cc(H, U)$ be the set of connected components in $H - U$ for any graph H and its vertex subset $U \subseteq V_H$, and $vcc(H, U) = \{V_C \mid C \in cc(H, U)\}$. The following lemma holds.

▶ **Lemma 5.** *Let $H_0, H_1, H_2, \ldots, H_{k-1}$ and $U_0, U_1, \ldots, U_{k-1}$ be respectively the sequences of subgraphs of G and their vertex subsets such that $H_i \in cc(H_{i-1}, U_{i-1})$ holds. Assuming k algorithms respectively enumerating the vertices in U_i for each $i \in [0, k-1]$, there exists a logspace algorithm of enumerating all the vertex subsets in $vcc(H_{k-1}, U_{k-1})$ using them as black-box subroutines.*

Proof. Since the straightforward recursive emulation of H_{k-1} takes the overhead exponential of k, such an approach applies only to the case of $k = O(1)$. Instead of emulating $H_{i-1} - U_{i-1}$ recursively, we use the emulation of $G - U$ for $U = \bigcup_{0 \leq i \leq k-1} U_i$. Since we assume the algorithms of enumerating U_i for all $i \in [0, k-1]$, this emulation works with a polynomial-time overhead independent of k. We have $cc(H_{k-1}, U_{k-1}) \subseteq cc(G, U)$ obviously. While

[5] Precisely, if two directed edges (u, v) and (v, u) exist, omitting their orientation causes two multiedges between u and v. Then those edges are merged into a single undirected edge.

$cc(G,U)$ might contain a connected component not in $cc(H_{k-1},U_{k-1})$, one can identify $C \in cc(H_{k-1},U_{k-1})$ by checking if C has an outgoing edge to a neighbor in U_{k-1} because any component $C \notin cc(H_{k-1},U_{k-1})$ is separated from H_{k-1} by $U_0, U_1, \dots$ or U_{k-2}. Thus, letting ∂U_{k-1} be the set of vertices in $G - \bigcup_{0\le i\le k-1} U_i$ adjacent to a node in U_{k-1}, it suffices to obtain an algorithm enumerating all the components in $cc(G,U)$ intersecting ∂U_{k-1}. It is realized by the following procedure.

1. Let $c = 1$, and $v_0, v_1, \dots, v_{l-1}$ be the sequence of the vertices in ∂U_{k-1} sorted by their IDs, which can be enumerated using logarithmic space and the algorithms for $U_0, U_1, \dots, U_{k-1}$.
2. For each v_i, check if a vertex v_j satisfying $j < i$ is reachable to v_i in $G - U$. If such a vertex exists, repeat this step for v_{i+1} (unless $i = l-1$). Otherwise, go to step 3.
3. Enumerate all the vertices reachable from v_i as the vertices in the c-th component in $cc(H_{k-1},U_{k-1})$. After the enumeration, increment c by one, and go back to step 2 for v_{i+1}.

The procedure above is implemented with $O(\log n)$-bit space by utilizing the logspace undirected s-t connectivity algorithm [41] (since $cc(H_{k-1},U_{k-1})$ is a set of weakly-connected components, undirected s-t connectivity suffices). Letting v_j be the vertex with the minimum ID in $V_C \cap \partial U_{K-1}$ for a component $C \in cc(H_{k-1},U_{k-1})$, C is necessarily enumerated when the procedure above processes v_j. In addition, it is never enumerated twice because any other vertex in $V_C \cap \partial U_{k-1}$ has an ID larger than v_j and is reachable to v_j in $G - U$. ◀

The lemma above implies that one can associate an unique integer ID with each connected component in $H_{k-1} - U_{k-1}$, and can emulate with a polynomial-time overhead the connected component in $H_{k-1} - U_{k-1}$ specified by a given ID. We also have the lemma below.

▶ **Lemma 6.** *Let $G = (V,E)$ be any graph of treewidth w, and $0 < \delta < 1$ be an arbitrary positive constant. Assume an algorithm outputting a tree decomposition of width at most w' for G. Then, there exists an algorithm outputting an $O(n^{-\delta})$-balanced vertex separator U of size $O(w'n^{\delta})$ for G, which uses only $O(w'n^{\delta}\log n)$-bit space (except for the space used by the tree-decomposition algorithm).*

Proof. Let $(\mathcal{T}, \{B_x\}_{x\in V_{\mathcal{T}}})$ be the (rooted) tree decomposition constructed by the algorithm. For any subgraph $\mathcal{T}' \subseteq \mathcal{T}$ and a subset $X \subseteq V_G$, we define $\mathrm{vol}(\mathcal{T}', X) = |\bigcup_{y\in V_{\mathcal{T}'}} B_y \setminus X|$. We also define $\mathcal{T}(x)$ as the subtree of $\mathcal{T}$ rooted by $x \in V_{\mathcal{T}}$.

The proof is constructive. The algorithm finds the $O(n^{\delta})$ bags whose removal splits $\mathcal{T}$ into a small subtrees $\mathcal{T}_0, \mathcal{T}_1, \dots, \mathcal{T}_{M-1}$ satisfying $\mathrm{vol}(\mathcal{T}_i, U) \le n^{1-\delta}$ for any $i \in [1, M]$. The algorithm manages two sets U' and U. The set U' stores the set of bag IDs constituting the vertex subset U, i.e., $U = \bigcup_{x\in U'} B_x$. The construction of U is done by iteratively adding a vertex in $V_{\mathcal{T}}$ to U'. Let x_i be the vertex added to U' at the i-th iteration, and U_i and U'_i be respectively the contents of U and U' when $|U'| = i$ holds. We further define $\mathcal{T}_i^R$ as the connected component of $\mathcal{T} - U'_i$ containing the root. The algorithm chooses as x_i the deepest vertex in $\mathcal{T}_{i-1}^R$ such that $\mathrm{vol}(\mathcal{T}_{i-1}^R(x_i), U_{i-1}) \ge n^{1-\delta}$ holds. The iteration terminates when $\mathrm{vol}(\mathcal{T}_i^R, U_i)$ becomes smaller than $n^{1-\delta}$. Since one iteration decreases $\mathrm{vol}(\mathcal{T}_*^R, U_*)$ by at least $n^{1-\delta}$, the algorithm terminates within n^{δ} iterations. In addition, for any children y of x_i, we have $\mathrm{vol}(\mathcal{T}_{i-1}^R(y), U_{i-1}) < n^{1-\delta}$ because x_i is the deepest vertex. It implies that any connected component in $G - U$ contains at most $n^{1-\delta}$ vertices.

The remaining issue is the time and space complexities for implementing the algorithm. Since the tree decomposition algorithm provides the whole topological information on $\mathcal{T}$, one can use it as the adjacency list of $\mathcal{T}$ incurring a polynomial-time overhead. Since U' is stored in the working memory, it is possible to emulate $\mathcal{T}_i^R$ for any i. The computation

of $\mathrm{vol}(\mathcal{T}_i^R(x), U_i)$ for any $x \in V_{\mathcal{T}_i^R}$ can be done in a polynomial time using the membership test of $v \in B_y$ for all pairs of $v \in V_G \setminus U_i$ and $y \in V_{\mathcal{T}_i^R(x)} \setminus \{x\}$. Consequently, the proposed algorithm can be implemented with the storage cost for U and U'. Since each bag contains at most w' vertices, the space complexity is $O(w'n^\delta \log n)$ bits. ◀

3 Small-Space Lex-DFS Algorithm for Graphs of Bounded Treewidth

3.1 Reduction to (Approximate) Gray-Path Membership

The algorithms shown in [3] reduces the procedures of $\textsc{Pivot}(t)$ and $\textsc{Parent}(t)$ to a single abstract task called $\textsc{IsGray}(u, v)$, which tests if v belongs to the prefix of the gray path by u (i.e., tests if $v \in V_{S_{d(u)}}$ holds or not). The following lemma is proved in [3].

▶ **Lemma 7** (Asano et al. [3]). *Assume that there exists a polynomial-time algorithm $\textsc{IsGray}(u, v)$ which is executable at any time $d(u) \leq t \leq l(u)$ and determines if $v \in V_{S_{d(u)}}$ holds or not. Letting $f(n)$ be the space complexity of that algorithm and $g(n)$ be the space-complexity of solving the directed s-t reachability problem, we have two polynomial-time algorithms which respectively implement $\textsc{Pivot}(t)$ and $\textsc{Parent}(t)$ using the memory of $f(n) + g(n)$ bits.*

We slightly extend this lemma by introducing a new primitive called $\textsc{AIsGray}(u, v)$ (approximate testing of gray vertex), which is a one-sided-error version of $\textsc{IsGray}(u, v)$ satisfying the following two conditions:

1. If $v \in V_{S_{d(u)}}$ holds, $\textsc{AIsGray}(u, v)$ always returns true.
2. If $d(v) > d(u)$ holds, $\textsc{AIsGray}(u, v)$ always returns false.

The following lemma implies that we can replace $\textsc{IsGray}(u, v)$ in Lemma 7 by $\textsc{AIsGray}(u, v)$.

▶ **Lemma 8.** *Assume a polynomial-time algorithm $\textsc{AIsGray}(u, v)$ executable at any time $d(u) \leq t \leq l(u)$ using $f(n)$-bit space, and a polynomial-time tree decomposition algorithm outputting the decomposition of width at most w' for any input graphs of treewidth w. Then we have the polynomial-time algorithm which implements $\textsc{IsGray}(u, v)$ using the space of $f(n) + O(w' \log n)$ bits (except for the space used by the tree decomposition algorithm).*

Proof. To implement $\textsc{IsGray}(u, v)$, it suffices to enumerate all the vertices in $S_t(u)$, which can be realized by repeatedly using an algorithm outputting $s_t(x)$ for given $x \in V_{S_t(u)} \setminus \{h_t\}$. Let $V'(x)$ be the set of the vertices v' such that $\textsc{AIsGray}(x, v')$ returns true. It has been shown in [3] that $s_t(x)$ is the first vertex $y \in N(x)$ with respect to the order of A_x such that y is reachable to h_t in $G - V_{S_t(x)}$. We first show that this fact still holds even if we replace $V_{S_t(x)}$ by $V'(x)$. Any y preceding $s_t(x)$ in A_x is unreachable to h_t in $G - V'(x)$ because it is unreachable in $G - V_{S_t(x)}$ and $V_{S_t(x)} \subseteq V'(x)$ holds by the first condition of $\textsc{AIsGray}$. Letting X be the graph corresponding to the suffix of S_t from $s_t(x)$ to h_t, any vertex in V_X has a discovery time larger than $d(x)$, and thus $V_X \cap V'(x) = \emptyset$ holds by the second condition of $\textsc{AIsGray}$. It implies that $s_t(x)$ is reachable to h_t in $G - V'(x)$, and concludes that $s_t(x)$ is the first vertex $y \in N(x)$ such that y is reachable to h_t in $G - V'(x)$. Since the procedure $\textsc{AIsGray}(x, v')$ works as the emulator of $G - V'(x)$, we can obtain an algorithm of computing $s_t(x)$ using any directed s-t reachability algorithm. The algorithm by Jain et al. [29] matches our goal. It uses any tree decomposition of width w' as a side information, and runs in a polynomial time using $O(w' \log n)$-bit space. The memory complexity is $f(n)$ bits for $\textsc{AIsGray}$, and $O(w' \log n)$ bits for deciding s-t reachability and for managing a constant number of pointers to vertices in V_G. ◀

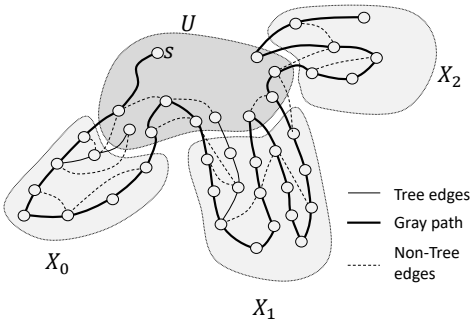

Figure 1 An example of decomposition by U.

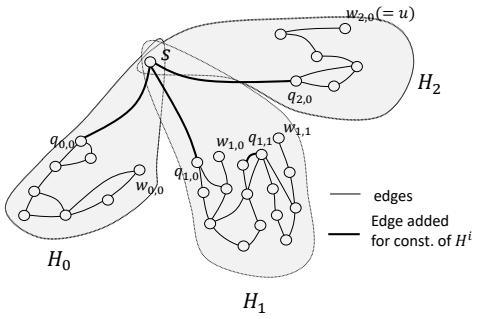

Figure 2 Construction of H_i.

3.2 Implementation of AIsGray(u, v)

As utilized in the proof of Lemma 8, directed s-t reachability is solvable using $O(w' \log n)$-bit space with the side information of a tree decomposition of width w' [29]. Thus we have Lemma 7 with $g(n) = O(w' \log n)$. The remaining part of our algorithm is to implement AIsGray(u, v) executable at any $d(u) \leq t \leq l(u)$. Let U be the set shown in Lemma 6, and $\mathcal{X} = \{X_0, X_1, \ldots, X_{N-1}\}$ be the set of the connected components in $G - U$ (Figure 1). At each time t, our algorithm keeps track of the information of $(d(x), p_t(x), s_t(x))$ for any $x \in V_{S_t} \cap U$. Specifically, we prepare the dictionary Z which maps any vertex x in U to the corresponding triple $(d(x), p_t(x), s_t(x))$ if $x \in S_t$ holds. We refer to the three elements in the triple for x as $Z[x].d$, $Z[x].p$, and $Z[x].s$ respectively. The contents of Z is updated in the main routine of Lex-DFS when the search head moves. Let ℓ_i be the number of connected components in the subgraph $S_{d(u)}[V_{X_i}]$. For each connected component C in $S_{d(u)}[V_{X_i}]$, we define its *entrance* and *exit* as the vertices with the minimum and maximum discovery times in V_C respectively. Let $\mathcal{C}_i = C_{i,0}, C_{i,1}, \ldots, C_{i,\ell_i-1}$ be the sequence of the connected components in $S_{d(u)}[V_{X_i}]$ sorted by the discovery times of their entrances. We also define the exit of $C_{i,-1}$ as s, which works as a sentinel value. Let $Q_i = (q_{i,0}, q_{i,1}, \ldots, q_{i,\ell_i-1})$ and $W_i = (w_{i,-1}, w_{i,0}, w_{i,1}, \ldots, w_{i,\ell_i-1})$ be the sequences of the entrances and exits associated with each component in $\mathcal{C}_i$ respectively. Now we construct the graph H_i from G by the following procedure:

1. Remove all the vertices not in $V_{X_i} \cup \{s\}$ as well as their incident edges.
2. For all $0 \leq j \leq \ell_i - 1$, contract the gray path from $w_{i,j-1}$ to $q_{i,j}$ into an edge. The positions of $q_{i,j}$ in $A_{w_{i,j-1}}$ and $w_{i,j-1}$ in $A_{q_{i,j}}$ are equal to those of $s_t(w_{i,j-1})$ and $p_t(q_{i,j})$ respectively.

Note that $s = w_{i,-1} = q_{i,0}$ holds if $s \in V_{X_i}$. We illustrate an example of the construction in Figure 2. Consider the run of any Lex-DFS algorithm in H_i until the discovery of w_{i,ℓ_i-1}, which outputs a vertex set $L_i \subseteq V_{X_i} \cup \{s\}$. Let $L(u) = L_0 \cup L_1 \cup \cdots \cup L_{N-1} \cup (S_{d(u)} \cap U)$. An important fact is that AIsGray(v, u) can be implemented using the query if $v \in L(u)$ or not. The following lemma holds.

▶ **Lemma 9.** *Let L_i be the output sequence of the Lex-DFS running in H_i until the discovery of w_{i,ℓ_i-1}. Any vertex in $V_{S_{d(u)}[V_{X_i}]}$ is contained in L_i, and $d(x) \leq d(w_{i,\ell_i-1}) \leq d(u)$ holds for any $x \in L_i$.*

Proof. For any $v \in V$, let $\mathcal{P}_{G,v}$ be the set of all simple paths from s to v in G, and $\mathcal{P}_G = \bigcup_{v \in V_G} \mathcal{P}_{G,v}$. For any path $P = s, u_1, \ldots, u_j, v$ in $\mathcal{P}_{G,v}$, we define its *word* $\gamma_G(P)$ as the sequence $A_s^{-1}[u_1], A_{u_1}^{-1}[u_2], \ldots, A_{u_k}^{-1}[v]$. Letting $\prec$ be the lexicographic order over all words, the *minimum path* $\pi_G(v) \in \mathcal{P}_{G,v}$ of a vertex $v \in V$ is defined as the one satisfying

$\gamma_G(\pi_G(v)) \prec \gamma_G(P)$ for any $P \in \mathcal{P}_{G,v}$. For any two vertices $v, v' \in V_{H_i}$, the gray paths in H_i to v and v' are respectively obtained by contracting several common subpaths in the gray paths to v and v' in G. Hence it is easy to check $\gamma_G(\pi_G(v)) \prec \gamma_G(\pi_G(v'))$ holds if and only if $\gamma_{H_i}(\pi_{H_i}(v)) \prec \gamma_{H_i}(\pi_{H_i}(v'))$ holds for any $v, v' \in V_{H_i}$. Since it is well-known that the total ordering of V_G with respect to $\prec$ over $\{\gamma_G(\pi_G(v))\}_{v \in V_G}$ is equivalent to the Lex-DFS ordering of V_G, this fact implies that L_i contains all the vertices in V_{X_i} discovered earlier than $w_{i,\ell-1}$ in the Lex-DFS search in G, and contains no vertex in V_{X_i} whose discovery time in the Lex-DFS search in G is later than $d(w_{i,\ell-1})$. The lemma is proved. ◀

Since H_i is a minor of G, its treewidth is also bounded by w. Thus we can perform our Lex-DFS algorithm recursively for graph H_i of $O(n^{1-\epsilon})$ vertices to output L_i. The graph H_i can be emulated using the subset U and the information stored in Z. Outputting L_i for all $i \in [0, N-1]$ can answer the query if $v \in L(u)$ holds or not.

3.3 Algorithm Details for Lex-DFS

The pseudocode of our algorithm is given in Algorithm 2. It is defined as a recursive procedure Lex-DFS(G, s, u), which outputs the Lex-DFS sequence of G starting from s until u is discovered. If the procedure runs with $u \notin V_G$, it outputs the whole Lex-DFS sequence of G starting from s. Note that the dictionary Z is independently defined in each recursive call for the computed separator U. In addition, the size $O(n^\epsilon)$ of separator U is fixed independently of recursion depth. That is, the variable n in the size parameter $O(n^\epsilon)$ is always the number of vertices in the original input graph, not the number of vertices in the input graph taken as an argument of Lex-DFS. The main routine Lex-DFS almost follows Algorithm 1, except for using AIsGray to compute Pivot and Parent and managing Z. The core of the algorithm is the implementation of AIsGray, in particular, the emulation of H_i for each V_i $(0 \leq i \leq N-1)$. That part corresponds to the lines 23-30. First, we identify the set Q_i and W_i, which can be done by extracting the nodes $x \in U$ satisfying $Z[x].s \in V_i$ as a member of Q_i (or those satisfying $Z[x].p \in V_i$ as W_i). Since each node in $S_t(u) \cap U$ stores its discovery time in Z, we can add the nodes into Q_i or W_i in the order of their discovery times. Following the order of Q_i and W_i, we create the edge set F, which corresponds to the edges obtained by the contraction of gray subpaths in the step 2 of the construction.

3.4 Complexity

Since Pivot and Parent are called at most $2n$ times in each recursive call, a polynomial-time invocations of AIsGray suffices to implement them. Let n^c be the upper bound for the number of invocations of AIsGray in one execution of Pivot or Parent. One invocation of AIsGray calls Lex-DFS at most n times. Putting all them together, n^{c+2} recursive invocations of Lex-DFS occur per one call of Lex-DFS. Since the recursion depth is obviously bounded by $O(1/\epsilon)$, at most $O(n^{(c+2)/\epsilon})$ calls of Lex-DFS are invoked in total. By Lemma 5, the input graph to each recursive call can be emulated with a polynomial-time overhead, and thus one invocation of Lex-DFS excepting the run of recursive calls has a polynomially-bounded running time. Consequently, the total running time is $n^{O(1/\epsilon)}$.

In each recursive call, the information on U and Z is stored in the working memory. The space for storing U and Z are bounded by $O(w' n^\epsilon \log n)$ bits. Except for the space used by the tree decomposition algorithm, the space of $O(w' \log n)$ bits is necessary for implementing Parent and Pivot from AIsGray. Since the recursion depth is $O(1/\epsilon)$, the total space complexity is $O(w' \epsilon^{-1} n^\epsilon \log n)$ bits.

Algorithm 2 Lex-DFS Algorithm for graph G of treewidth k (starting from s).

function LEX-DFS(G, s, u)
  if $|V_G| \leq n^\epsilon$ **then** run the standard Lex-DFS algorithm and halt
  Find a separator U of size $O(n^\epsilon)$
  Initialize $Z\colon U \to [0, n-1] \times V_G \times V_G$
  $v_{cur} \leftarrow s$; $t \leftarrow 1$
  output s; $Z[s] \leftarrow (1, \bot, \bot)$
  while true **do**
    $v \leftarrow$ PIVOT(t) using AISGRAY($v_{cur}, \cdot$)
    if $v = -1$ **then** ▷ All neighbors have been already visited
      $v \leftarrow$ PARENT(t) using AISGRAY($v_{cur}, \cdot$)
      $Z[v_{cur}] \leftarrow$ null; $Z[v] \leftarrow (Z[v].d, Z[v].p, \bot)$
      if $v = -1$ **then** halt ▷ All vertices are visited
    else
      $Z[v_{cur}] \leftarrow (Z[v_{cur}].d, Z[v_{cur}].p, v)$; $Z[v] \leftarrow (t, v_{cur}, \bot)$
      Output v
      if $v = u$ **then** halt
    $v_{cur} \leftarrow v$
    $t \leftarrow t + 1$
  function AISGRAY(u, v)
    if $Z[v] \neq$ null **then** return true
    Let $X_0, X_1, \ldots, X_{N-1}$ be the connected components in $G - U$
    for $i = 0, 1, \ldots, N-1$ **do**
      $Q \leftarrow ()$; $W \leftarrow (s)$
      for $\forall x \in U$: $Z[x] \neq$ null in ascending order of $Z[x].d$ **do**
        if $Z[x].s \in V_{X_i}$ **then** append $Z[x].s$ to Q
        if $Z[x].p \in V_{X_i}$ **then** append $Z[x].p$ to W
      if $u \in V_{X_i}$ **then** append u to W
      $\ell_i \leftarrow |Q|$
      for $j = 0, 1, \ldots, \ell_i - 1$ **do**
        $F \leftarrow F \cup \{(W[j], Q[j])\}$
      $H_i \leftarrow G[V_i] + F$ ▷ Not explicitly constructed
      if $v \in$ LEX-DFS($H_i, s, W[\ell_i - 1]$) **then** return true
    return false

4 Tree Decomposition using Small Space

In this section, we present a tree-decomposition algorithm, which uses $O(wn^{1/2} \log n)$-bit space and outputs a decomposition of width $O(wn^{1/2} \log n)$ for any undirected graph $G = (V, E)$ of treewidth $w \leq \sqrt{n}$. We first introduce a space-saving variant of the known weighted balanced separator algorithm.

▶ **Lemma 10** (Extended from Theorem 1.1 of Fomin et al. [24]). *There exists a polynomial-time algorithm that, given a graph G on n vertices, any vertex-weight function μ, and a positive integer k, either provides a weighted $O(1)$-balanced separator of G with respect to μ consisting*

of $O(k^2)$ vertices, or concludes that the treewidth of G is more than k. The algorithm uses the memory space required for finding the minimum unweighted s-t vertex cut in G plus $O(k^2 \log n)$ bits.

Proof. We refer to the algorithm proposed in Theorem 1.1 of [24] as Sep. Except for the space complexity matter, the correctness of the lemma completely follows that of Sep presented in [24]. Thus it suffices to show how Sep is implemented using the memory space claimed in this lemma. The algorithm Sep roughly works as follows. Let G be any input graph of treewidth at most k.

1. First the algorithm Sep constructs any rooted spanning tree T of G, and decomposes it into a set $\mathcal{X}$ of $\Theta(k)$ connected subtrees such that at most $O(k)$ vertices can belong to two or more subtrees in $\mathcal{X}$: Let $T(x)$ be the subtree of T rooted by x. The algorithm Sep starts with $T' = T$, and for $i = 0, 1, \ldots$, iteratively finds the deepest vertex x_i such that $|V_{T'(x_i)}| \geq \alpha n/k$ holds for an appropriate constant α. Then the subtree $T'(x_i)$ is split into several subtrees of size $\Theta(n/k)$ sharing x_i, each of which becomes a member of $\mathcal{X}$. After updating T' as $T' \leftarrow T' - T'(x_i)$, the algorithm proceeds to the next iteration.
2. For any $X_i, X_j \in \mathcal{X}$ such that $V_{X_i} \cap V_{X_j} = \emptyset$, Sep emulates the graph $H_{i,j}$ obtained from G by contracting X_i and X_j into two vertices x_i and x_j. Then it computes the minimum x_i-x_j vertex cut in $H_{i,j}$. If there exists a pair (i, j) such that the output cut contains at most k vertices, the algorithm adds it to the separator set U.
3. The steps 1 and 2 are iteratively applied to the largest connected component after the removal of the computed vertex cut, until U becomes an $O(1)$-balanced separator of G. It is proved in [24] that this iteration terminates within $O(k)$ times if the treewidth of the input graph is at most k.

The small-space implementation of step 1 is very similar with the algorithm shown in the proof of Lemma 6. With the support of the emulator of T, finding x_i and the emulation of T' can be done in the same way as the proof of Lemma 6. The spanning tree T can be emulated using the logspace undirected connectivity: We introduce an arbitrary logspace-computable edge-weight function $g\colon E_G \to \mathbb{N}$ which assigns all edges with different weights. Let $e_0, e_1, \ldots, e_{m-1}$ be the sequence of all edges sorted in the ascending order of their weights, and $E_i = \{e_0, e_1, \ldots, e_{i-1}\}$. Then an edge $e_i = (u, v)$ is contained in the minimum spanning tree of G with respect to g if and only if u and v is connected in $G[E_i]$, which directly deduces the emulator of the minimum spanning tree.

Assuming an algorithm computing the set $\mathcal{X}$, $H_{i,j}$ can be emulated with a polynomial-time overhead. Thus the step 2 can be implemented within a polynomial time using $O(k^2 \log n)$ bits (except for the space used by the vertex-cut algorithm). ◀

4.1 A Small-Space Balanced Separator Algorithm

Let $I(G)$ be the maximum independent set of G, (if two or more maximum independent sets exist, an arbitrary one is chosen), and $\overline{I}(G) = V \setminus I(G)$ for short. The first key ingredient of our algorithm is a space-saving algorithm for the minimum s-t vertex cut problem.

▶ **Lemma 11.** *Let G be any n-vertex undirected graph. For any $s, t \in V_G$, the minimum (unweighted) s-t vertex cut of G can be found in a polynomial time using $O(|\overline{I}(G)| \log n)$ bits.*

Proof. The algorithm basically follows the vertex-cut version of Ford-Fulkerson algorithm, which manages a set of augmenting paths for recognizing the current residual graph (see Section 3.5 in [35] for example). In the case of unweighted vertex cuts, any set of augmenting paths is a set of vertex-disjoint s-t paths in G. Letting L be the maximum total length of

managed s-t paths, the algorithm can be implemented using $O(L \log n)$-bit space. Thus it suffices to show that $L = O(|\overline{I}(G)|)$ holds for any instance G. Let $R_1, R_2, \ldots, R_y$ be any set of vertex-disjoint s-t paths in G. Since no two vertices in $I(G)$ consecutively appears in any path, we have $|V_{R_i} \cap I(G)| \leq |V_{R_i} \cap \overline{I}(G)| + 1$ for any $i \in [1, y]$. Then $|V_{R_i}| \leq 3|V_{R_i} \cap \overline{I}(G)|$ holds. Since V_{R_i} for all $i \in [1, y]$ are mutually disjoint, it follows $\sum_{1 \leq i \leq y} |V_{R_i}| = O(|\overline{I}(G)|)$. The lemma is proved. ◀

Consider a partition of V_G into a family $\mathcal{P} = \{P_0, P_2, \ldots, P_{N-1}\}$ of N subsets such that $G[P_i]$ is a connected subgraph of G. We denote by $G/\mathcal{P}$ the graph obtained by contracting each subgraph P_i into a single vertex u_i with weight $\mu(u_i) = |V_{P_i}|$ (parallel edges are merged into the single one). The second key ingredient is to reduce the (approximate) tree decomposition of G into that of another graph $G/\mathcal{P}$ for an appropriate partition $\mathcal{P}$ such that $|\overline{I}(G/\mathcal{P})|$ becomes small. Since treewidth never increases by edge contraction, $G/\mathcal{P}$ also has a treewidth at most k. Thus we can run the balanced-separator algorithm obtained from Lemmas 10 and 11 on $G/\mathcal{P}$ using only $O((|\overline{I}(G/\mathcal{P})| + k^2) \log n)$-bit space. For the computed separator B, we replace each $u_i \in B$ by V_{P_i}. That is, we create a vertex subset $B' = \bigcup_{u_i \in B} V_{P_i}$, which is obviously a balanced separator of G consisting of $O(k^2 \max_i\{|V_{P_i}|\})$ vertices. To attain the space complexity of Theorem 2 following this approach, we have to construct a polynomial-time algorithm outputting the partition $\mathcal{P}$ satisfying $|\overline{I}(G/\mathcal{P})| = O(kn^{1/2})$ and $|P_i| = O(n^{1/2}/k)$ for any $P_i \in \mathcal{P}$. In addition, we have to guarantee that the algorithm uses only $O(kn^{1/2} \log n)$ bits.

We argue the implementation of such an algorithm. Let us define an arbitrary total ordering of edges in E_G. This can be easily realized by any ordering function $f : E \to \mathbb{N}$ which can be computed in logarithmic space (e.g., the lexicographic ordering of endpoint ID pairs). Let $e_1, e_2, \ldots, e_m$ be the sequence of all edges sorted in this order, and $E_i = \{e_1, e_2, \ldots, e_i\}$. For any subgraph $H \subseteq G$, we also define $S(H, v, i)$ as the set of the vertices which are reachable from v in $H[E_i]$. The partition is constructed by the following algorithm:

1. Let $\mathcal{P} = \emptyset$, $R \leftarrow \emptyset$, and $H \leftarrow G$.
2. Find the minimum ℓ such that the largest connected component C in $H[E_\ell]$ contains at least $n^{1/2}/k$ vertices, and add V_C to $\mathcal{P}$. Letting v be the vertex with the smallest ID in V_C, we store the pair (v, ℓ) into R.
3. Update $H \leftarrow G - \bigcup_{(v,\ell) \in R} S(G, v, \ell)$.
4. Repeat steps 2 and 3 until the size of any connected component in H becomes less than $n^{1/2}/k$.
5. Letting $Q = \bigcup_{P_i \in \mathcal{P}} P_i$, add $V_{C'}$ to $\mathcal{P}$ for any connected component C' in $G - Q$.

The actual algorithm does not store $\mathcal{P}$ explicitly. Except for step 5, the set $\mathcal{P}$ is write-only, and it is easy to verify that steps 1-4 can be implemented only with the space for storing R. The matter of the space complexity relies on how to restore the set Q in step 5 only from the information of R. Let C_i be the connected component found in the i-th iteration of step 2, and (u_i, j_i) be the entries added to R then. We denote by H_i the graph stored in H immediately after the i-th iteration of step 2. It is easy to enumerate the vertices in $S(G, u, j)$ by the logspace undirected s-t connectivity algorithm [41] (recall that we omit the orientation of edges in considering the tree decomposition for directed graphs), and thus we obtain an emulator of H_i using the information in R and extra $O(\log n)$-bit space. It also yields an algorithm for enumerating $C_i = S(H_{i-1}, u_i, j_i)$. Consequently, this algorithm works using only $O(|R| \log n)$-bit space. The following lemma guarantees the correctness of the output.

▶ **Lemma 12.** *Let $\mathcal{P} = \{P_0, P_1, \ldots, P_{N-1}\}$ be the partition outputted by the algorithm above. Then for any $P_i \in \mathcal{P}$, $|P_i| \leq 2n^{1/2}/k$ holds. In addition, $|\overline{I}(G/\mathcal{P})| \leq kn^{1/2}$ holds.*

Proof. We first show that $\mathcal{P}$ is actually a partition of V_G. By step 5, it is obvious that any vertex in V_G is contained in at least one subset $P_i \in \mathcal{P}$. The subset added in step 5 does not intersect other subsets. Consider any two subsets C_i and C_k added in step 2 ($i < k$). By the construction of C_i and C_k, we have $C_i = S(H_{i-1}, u_i, j_i)$ and $C_k = S(H_{k-1}, u_k, j_k)$. Since $i < k$ holds, H_{k-1} does not contain any vertex in $S(G, u_i, j_i)$. It implies H_{k-1} does not contain any vertex in $C_i = S(H_{i-1}, u_j, j_i)$ because of $H_{i-1} \subseteq G$. That is, C_i and C_k are mutually disjoint. Next, we bound the size of each P_i. Any subset added in step 5 has a cardinality less than $n^{1/2}/k$. Since the algorithm finds the smallest ℓ such that the cardinality of C becomes at least $n^{1/2}/k$, any connected component in $H_{i-1}[E_{\ell-1}]$ has a size less than $n^{1/2}/k$. Thus the size of any connected component in $H_{i-1}[E_\ell]$ is at most $2n^{1/2}/k$. Finally, we show $|\overline{I}(G/\mathcal{P})| \leq kn^{1/2}$. We call a subset $P_i \in \mathcal{P}$ a *red subset* if P_i is added in step 5, and also call the corresponding vertex $u_i \in V_{G/\mathcal{P}}$ a *red vertex*. It is obvious that any red subset P_i has no outgoing edge to other red subsets in G, the set of all red vertices forms an independent set of $G/\mathcal{P}$. Since the cardinality of any non-red subset is at least $n^{1/2}/k$, at most $kn^{1/2}$ non-red vertices exist in $G/\mathcal{P}$. It implies $\overline{I}(G/\mathcal{P}) \leq kn^{1/2}$. The lemma is proved. ◀

This lemma also implies that the size of R is at most $kn^{1/2}$. Thus the space complexity of the algorithm is bounded by $O(kn^{1/2} \log n)$ bits. The combination of Lemmas 10, 11, and 12 yields the following lemma.

▶ **Lemma 13.** *There exists a polynomial-time algorithm that, given a graph G on n vertices and a positive integer k, either provides an $O(1)$-balanced separator of G consisting of $O(kn^{1/2})$ vertices, or concludes that the treewidth of G is more than k. The algorithm uses $O(kn^{1/2} \log n)$-bit space.*

The remaining part is to transform the separator algorithm into a tree-decomposition algorithm. The following lemma obviously deduces Theorem 2.

▶ **Lemma 14.** *Assume an algorithm Alg which outputs an $O(1)$-balanced separator of size $k(w, n)$ for any graph G of treewidth w using $g(n)$-bit space. Then there exists a polynomial-time tree decomposition algorithm which outputs a tree decomposition of width $O(k(w, n) \log n)$ using $O(g(n) + k(w, n) \log^2 n)$-bit working memory.*

Proof. The proof is constructive. We refer to the constructed algorithm as *Alg*. The algorithm *Alg* first computes an $O(1)$-balanced separator U of G, and then recursively constructs the tree decomposition of each connected component in $G - U$ whose size is larger than $k(w, n)$. A component having at most $k(w, n)$ vertices is treated as the subgraph with the tree decomposition consisting of a single bag of the whole component. Let $H_0, H_1, \ldots, H_{\ell-1}$ be the connected components in $G - U$ sorted in the order specified by the enumeration algorithm of Lemma 5, and T_i be the output sequence of the recursive call for H_i. Defining the binary operator $\circ$ for concatenating two sequences, let $T = T_0 \circ T_1 \circ \cdots \circ T_{\ell-1}$. We further define $m_i = \sum_{0 \leq j \leq i} |T_i|$. The algorithm *Alg* modifies the sequence T in the following way: Any pair $(B, q) \in T_i$ is replaced by $(B \cup U, q + m_i)$ if $q \neq -1$ or $(B \cup U, m_{\ell-1})$ otherwise. Finally, we append the pair $(U, -1)$ at the tail of T. Intuitively, this modification is for relabeling bag identifiers to guarantee their uniqueness, and for merging all the subgraph decompositions into a single one rooted by the last bag $(U, -1)$. It is easy to verify that the modified sequence is a tree decomposition of G. By Lemma 5, the input graph at any recursion level is emulated with a polynomial-time overhead. Thus the running time at any recursion level is a polynomial time. One recursive call remove one bag from the input graph, the number of recursive calls is bounded by n. Totally the algorithm *Alg* finishes within a polynomial time.

Let w_i be the maximum width of all the output tree decompositions at the i-th recursion level. Then we have the inequality $w_{i+1} \leq k(w,n)+w_i$. Since the separator is $O(1)$-balanced, the recursion finishes at the depth of $O(\log n)$. It implies that the maximum bag size is $O(k(w,n)\log n)$. The modification of the sequence T can be done in a streaming way. Thus the space complexity of *Alg* is $O(|U|\log n) = O(k(w,n)\log n)$ bits per one recursion. The largest space consumption is at the bottom-level recursion, where *Alg* uses $O(k(w,n)\log^2 n)$ bits in total. ◀

The lemma above also deduces the consequence for planar graphs in Corollary 3. Since planar graphs admit a $\tilde{O}(\sqrt{n})$-bit space $O(1)$-balanced separator algorithm [27], we can use it instead of Lemma 13.

5 Conclusion

In this paper, we presented a Lex-DFS algorithm for directed graphs of bounded treewidth w. It is not only the first algorithm solving Lex-DFS using sublinear space for $w = \omega(1)$, but also the first algorithm solving directed s-t reachability in the same situation. One of the key tools is a new sublinear-space tree decomposition algorithm covering the case of moderate (small but non-constant) treewidth. The authors believe that this is a strong tool for designing small-space algorithms for other fundamental graph problems on bounded-treewidth graphs.

References

1. Eric Allender, David A. Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and grid graph reachability problems. *Theory of Computing Systems.*, 45(4):675–723, 2009.
2. Richard Anderson and Ernst W. Mayr. Parallelism and the maximal path problem. *Information Processing Letters*, 24(2):121–126, 1987. doi:10.1016/0020-0190(87)90105-0.
3. Tetsuo Asano, Taisuke Izumi, Masashi Kiyomi, Matsuo Konagaya, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, Jun Tarui, and Ryuhei Uehara. Depth-first search using $O(n)$ bits. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 553–564, 2014.
4. Tetsuo Asano, David Kirkpatrick, Kotaro Nakagawa, and Osamu Watanabe. $\widetilde{O}(\sqrt{n})$-space and polynomial-time algorithm for planar directed graph reachability. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 45–56, 2014.
5. Ryo Ashida and Kotaro Nakagawa. $\widetilde{O}(n^{1/3})$-space algorithm for the grid graph reachability problem. In *International Symposium on Computational Geometry (SoCG)*, pages 5:1–5:13, 2018.
6. Niranka Banerjee, Sankardeep Chakraborty, and Venkatesh Raman. Improved space efficient algorithms for BFS, DFS and applications. In *International Computing and Combinatorics Conference (COCOON)*, pages 119–130, 2016.
7. Bahareh Banyassady, Matias Korman, Wolfgang Mulzer, André van Renssen, Marcel Roeloffzen, Paul Seiferth, and Yannik Stein. Improved Time-Space Trade-Offs for Computing Voronoi Diagrams. In *International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 9:1–9:14, 2017. doi:10.4230/LIPIcs.STACS.2017.9.
8. Luis Barba, Matias Korman, Stefan Langerman, Kunihiko Sadakane, and Rodrigo I. Silveira. Space-time trade-offs for stack-based algorithms. *Algorithmica*, 72(4):1097–1129, 2015. doi:10.1007/s00453-014-9893-5.
9. Greg Barnes, Jonathan F. Buss, Walter L. Ruzzo, and Baruch Schieber. A sublinear space, polynomial time algorithm for directed s-t connectivity. *SIAM Journal on Computing*, 27(5):1273–1282, 1998. doi:10.1137/S0097539793283151.

10 Allan B. Borodin and Stephan Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM Journal on Computing*, 11(2):287–297, 1982. doi:10.1137/0211022.

11 Harry Buhrman, Richard Cleve, Michal Koucký, Bruno Loff, and Florian Speelman. Computing with a full memory: Catalytic space. In *ACM Symposium on Theory of Computing (STOC)*, pages 857–866, 2014. doi:10.1145/2591796.2591874.

12 Diptarka Chakraborty, A. Pavan, Raghunath Tewari, N. Variyam Vinodchandran, and Lin Forrest Yang. New Time-Space Upperbounds for Directed Reachability in High-genus and H-minor-free Graphs. In *International Conference on Foundation of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 585–595, 2014.

13 Diptarka Chakraborty and Raghunath Tewari. An $O(n^\epsilon)$ space and polynomial time algorithm for reachability in directed layered planar graphs. *ACM Transactions on Computation Theory*, 9(4), 2017.

14 Sankardeep Chakraborty, Anish Mukherjee, Venkatesh Raman, and Srinivasa Rao Satti. A Framework for In-place Graph Algorithms. In *European Symposium on Algorithms (ESA)*, volume 112, pages 13:1–13:16, 2018. doi:10.4230/LIPIcs.ESA.2018.13.

15 Sankardeep Chakraborty, Venkatesh Raman, and Srinivasa Rao Satti. Biconnectivity, Chain Decomposition and st-Numbering Using O(n) Bits. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 64, pages 22:1–22:13, 2016. doi:10.4230/LIPIcs.ISAAC.2016.22.

16 Sankardeep Chakraborty and Srinivasa Rao Satti. Space-efficient algorithms for maximum cardinality search, its applications, and variants of bfs. *Jouanal of Combinatorial Optimization*, 37(2):465–481, 2019. doi:10.1007/s10878-018-0270-1.

17 Timothy M. Chan. Comparison-based time-space lower bounds for selection. *ACM Transactions on Algorithms*, 6(2):26:1–26:16, 2010. doi:10.1145/1721837.1721842.

18 Timothy M. Chan, J. Ian Munro, and Venkatesh Raman. Faster, space-efficient selection algorithms in read-only memory for integers. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 405–412, 2013. doi:10.1007/978-3-642-45030-3_38.

19 Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85(1):12–75, 1990. doi:10.1016/0890-5401(90)90043-H.

20 Omar Darwish and Amr Elmasry. Optimal time-space tradeoff for the 2D convex-hull problem. In *European Symposium on Algorithms (ESA)*, pages 284–295, 2014.

21 Jeff Edmonds, Chung Keung Poon, and Dimitris Achlioptas. Tight lower bounds for st-connectivity on the nnjag model. *SIAM Journal on Computing*, 28(6):2257–2284, 1999. doi:10.1137/S0097539795295948.

22 Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 143–152, 2010.

23 Amr Elmasry, Torben Hagerup, and Frank Kammer. Space-efficient Basic Graph Algorithms. In *International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 288–301, 2015. doi:10.4230/LIPIcs.STACS.2015.288.

24 Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, Michał Pilipczuk, and Marcin Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. *ACM Transactions on Algorithms*, 14(3), 2018.

25 Greg N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *Journal of Computer and System Sciences*, 34(1):19–26, 1987. doi:10.1016/0022-0000(87)90002-X.

26 Torben Hagerup. Space-efficient DFS and applications to connectivity problems: Simpler, leaner, faster. *Algorithmica*, 82(4):1033–1056, 2020. doi:10.1007/s00453-019-00629-x.

27 Tatsuya Imai, Kotaro Nakagawa, Aduri Pavan, N. Variyam Vinodchandran, and O. Watanabe. An $O(n^{1/2+\epsilon})$-space and polynomial-time algorithm for directed planar reachability. In *IEEE Conference on Computational Complexity (CCC)*, pages 277–286, 2013.

28 Rahul Jain and Raghunath Tewari. An $O(n^{1/4+\epsilon})$ Space and Polynomial Algorithm for Grid Graph Reachability. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 19:1–19:14, 2019.

29 Rahul Jain and Raghunath Tewari. Reachability in High Treewidth Graphs. In *International Symposium on Algorithms and Computation (ISAAC)*, volume 149, pages 12:1–12:14, 2019. doi:10.4230/LIPIcs.ISAAC.2019.12.

30 Frank Kammer, Dieter Kratsch, and Moritz Laudahn. Space-efficient biconnected components and recognition of outerplanar graphs. *Algorithmica*, 81(3):1180–1204, 2019. doi:10.1007/s00453-018-0464-z.

31 Frank Kammer, Johannes Meintrup, and Andrej Sajenko. Space-efficient vertex separators for treewidth. *CoRR*, abs/1907.00676, 2019. arXiv:1907.00676.

32 Frank Kammer and Andrej Sajenko. Linear-time in-place dfs and bfs on the word ram. In *International Conference on Algorithms and Complexity (CIAC)*, pages 286–298, 2019.

33 Shahbaz Khan and Shashank K. Mehta. Depth First Search in the Semi-streaming Model. In *International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 126, pages 42:1–42:16, 2019. doi:10.4230/LIPIcs.STACS.2019.42.

34 Masashi Kiyomi, Hirotaka Ono, Yota Otachi, Pascal Schweitzer, and Jun Tarui. Space-efficient algorithms for longest increasing subsequence. *Theory of Computing Systems*, 2019. doi:10.1007/s00224-018-09908-6.

35 Tom Leighton and Satish Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999. doi:10.1145/331524.331526.

36 Andrea Lincoln, Virginia Vassilevska Williams, Joshua R. Wang, and R. Ryan Williams. Deterministic Time-Space Trade-Offs for k-SUM. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 55, pages 58:1–58:14, 2016. doi:10.4230/LIPIcs.ICALP.2016.58.

37 J.Ian Munro and Mike Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12(3):315–323, 1980. doi:10.1016/0304-3975(80)90061-4.

38 Jakob Pagter and Theis Rauhe. Optimal time-space trade-offs for sorting. In *ACM Symposium on Theory of Computer Science (STOC)*, pages 264–268, 1998. doi:10.1109/SFCS.1998.743455.

39 Chung Keung Poon. Space bounds for graph connectivity problems on node-named jags and node-ordered jags. In *IEEE 34th Annual Symposium on Foundations of Computer Science (FOCS)*, page 218–227, 1993.

40 John H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985. doi:10.1016/0020-0190(85)90024-9.

41 Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4), 2008. doi:10.1145/1391289.1391291.

42 Joshua R. Wang. Space-efficient randomized algorithms for k-sum. In *European Symposium on Algorithms (ESA)*, pages 810–829, 2014.